WHY WOULD ANYONE NEED JAVASCRIPT GENERATOR FUNCTIONS?

Written by James Sinclair on the 7th November 2022

Generators are an *odd* part of the JavaScript language. And some people find them a bit of a puzzle. You might be a successful developer for decades and never feel the need to reach for them. Which raises the question, if you can go so long without ever needing them, what are they good for?

Generators have a funny syntax, too. They have these strange starred function definitions; you can't define them with arrow functions; they add this mysterious keyword. If you're not familiar with what they're doing, they can make code impossible to read.

Part of the trouble is that generators are a low-level construct. That is, they're kind of like a tool for building tools. And it's those tools we build that solve day-to-day problems. But if you're looking at generator functions in isolation, it can be hard to see why you'd ever want them.

Generators are quite a powerful construct, though. And they're handy to have in your metaphorical toolbox. Like most tools, you might not need them for every single job. But for certain jobs, they make life much easier. Once you understand the tool better, you begin to see where they're helpful. And, equally important, you begin to see when *not* to use them.

What, then, are generators good for?

TIM TAMS AND LAZY ITERATORS

Generators have *lots* of uses. But the most immediate and obvious application is to make *lazy iterators*.

Now, I'm hoping you're already familiar with JavaScript's *iteration protocols*. These protocols are another low-level language feature. They let us tell the JavaScript engine that some object can be used in a loop or with *spread syntax*.

The simplest thing we can do is define a generator function that yields some values. We can then use a loop to iterate over them:

And on the surface, that looks pointless. You could do all that with a lot less rigmarole using arrays. But, ironically, it's generator's similarity to arrays that makes them so useful.

To explain why I must introduce you to the pinnacle of Australia's cultural achievement. And no, it's not the invention of Wi-Fi. Nor is it the Sydney Opera House. And it's not even the superlative coffee. [1] Arguably, Australia's greatest cultural achievement is the *Tim Tam*.

A plate of Tim Tams, with one in the centre broken open to show the biscuit and chocolate cream filling. Photograph by Bilby.

A Tim Tam "consists of two malted biscuits separated by a light hard chocolate cream filling and coated in a thin layer of textured chocolate." [2] These "creamy bricks of chocolatey goodness" [3] will serve as an analogy for us. A Tim Tam represents an item of data that we wish to process.

And we 'process' a Tim Tam through a ritual known as the *Tim Tam Slam*. The steps are as follows:

1. Select a single Tim Tam.
2. Bite a small chunk from one corner, 2–5 mm from the apex.
3. Repeat the bite on the diagonally opposite corner.
4. Insert one of the bitten corners into a hot beverage. (*Milo* is traditional, but coffee, tea, or hot chocolate is also acceptable).
5. Place your lips over the opposite corner, and draw liquid through the Tim Tam as if it were a straw.
6. As soon as liquid enters your mouth, immediately consume the entire Tim Tam. It's important to do this quickly before it loses its structural integrity.
7. Repeat until there are no more Tim Tams, or you feel physically ill.

Here's Australian singer Natalie Imbruglia demonstrating on the *Graham Norton Show*:

Some people might find the Australian and British accents difficult to understand. If that's the case for you, *Neil deGrasse Tyson breaks it down in a more recent YouTube video*.

Now, suppose we can consume at most five Tim Tams before starting to feel ill. If we were to represent this process using JavaScript, it might look like the following:

We have described the process of the Tim Tam Slam as a logical sequence of steps. And this is a good thing. But, there are serious problems with how we've written this function. A standard packet of original Tim Tams contains eleven biscuits. If we processed the packet as if it were an array, we'd take out each biscuit, bite a corner off, and place it in a new packet. Then we'd take each biscuit in turn and bite off the diagonally opposite corner. And we'd have a packet of eleven biscuits with two corners bitten off. But it all becomes ridiculous when we attempt the next step. That would be where we insert eleven biscuits into our beverage.

Even if we were to rearrange the order so that we ran the operation first, we'd still have trouble. We'd end up with five biscuits in our beverage at once. We could also compose all the functions from the operations together (using _____). Combining that with moving to the start would improve the situation.

This approach isn't *too* bad. But it's not as clean and clear as our first attempt. But generators, being lazy, offer us a way to keep that neat sequencing from the first example. To make it work, though,

we need to define some utility functions. First, we'll define a counterpart for Array's method. It will look like this:

Note the syntax. It means that this function will return a generator. And the generator implements the *iterable protocol*. This means that we can take that return value and use it in another structure. And we can create another function that will limit the number of items:

With these two utility functions, we can attempt to solve our problem. (With perhaps a little help from):

We've now achieved a nice, neat description of the process, much like our array methods version. Because generators are lazy, we won't end up with five biscuits in a mug. Except, there is a problem with this function. In its current form, running won't do anything. This is because generators are so lazy, they won't do *anything*. That is, unless we take some action to pull values out. There are two common ways to do this:

1. Convert the generator object into an array using spread syntax; or
2. Iterate over the generator *without* yielding any values.

If we take the second approach, we can write a utility, much like the array method . It might look like so:

Notice that there's no asterisk or keyword in this utility. It just takes an effect and runs it on each value from the iterator. With that in place, if we add another imaginary function, , to our pipeline, we can make use of our new utility:

And now our function will slam five Tim Tams in the correct order. But it will still limit processing to five (that is,).

This feature of 'laziness' is the first thing that makes generator functions useful. It allows us to change the way in which we process data. We can, for example, process large data sets by loading one item at a time into memory. And, on its own, this is enough to make generators interesting. But laziness has other advantages.

INFINITE ITERATORS

Back in the 1990s, Arnotts ran a series of commercials for Tim Tams. The first of these featured Cate Blanchett meeting a genie. And involved her character wishing for a packet of Tim Tams that never runs out.

The *whole series* of ads focussed around the concept of an infinite packet of Tim Tams. And, much like a packet of Tim Tams that never runs out, generators can create infinite iterators.

We could, for example, create a generator function that gives us an infinite sequence of ones:

Now, that might be interesting. But perhaps not so useful. We could, though, make this a little more general by specifying the value we want to repeat:

Still, that may not seem so useful. But we could use this to build other sequences. Suppose we defined a function called . It works a little bit like Array's method. But, instead of returning a single value, it yields a sequence of values.

And using , we could produce the sequence of natural numbers:

Although to be fair, it's easier to write:

The point here isn't to show you the most effective way to generate a lazy list of positive integers. It's that tools like and allow us to build complex infinite sequences out of simple ones.

I'll admit, though, generating an infinite list of positive integers isn't all that interesting. Let's try something genuinely useful. For example, something that's useful for cryptography or creating the appearance of randomness. We'll generate a sequence of prime numbers.

To achieve this, we need two more helper functions. First, we will want to filter a generated sequence:

And our second utility function, we'll call :

This function reveals a weakness of generators (and iterators in general). The weakness is that they are mutable. Popping one item from the sequence means we can't get that original sequence anymore. This is something to be careful of as you're working with generators. For now, though, we have these two helper functions, so we can put our prime number generator together. And we'll do it using a technique called *the Sieve of Eratosthenes*. The algorithm works as follows:

1. Start with a list of all natural numbers, beginning from 2.
2. Take the first element in the list, then remove all multiples of that number from the list.
3. Go to step 2, and repeat with the next number in the list.

To make this work in JavaScript, we create two functions. One to do the work of sieving through the list of numbers, and another that kicks everything off. Our sieving function looks like so:

Note that keyword there. If we have an iterable object, says 'yield everything from this other iterable.' This allows us to do something much like for arrays.

Now, with our sieve function ready to go, we need to kick it off with the list of all natural numbers starting from two. We can generate that with help from one more utility function:

And we can now put it all together:

Of course, there are other, more efficient ways to generate prime numbers (probably). But the point here is not about prime numbers. Rather, it's to show how infinite sequences allow us to think differently about a problem. We can also use laziness and infinite sequences for applications like:

- Generating a series of unique identifiers;
- Generating all the possible moves in a game; or
- Seeking a particular value (or values) amongst a bunch of permutations and combinations.

WHY AREN'T ALL THESE UTILITY FUNCTIONS BUILT-IN?

Through all the examples so far, we've been writing our own little utility functions. These include functions like , , , , , and . And it *is* a little annoying that generators don't have built-in methods like Arrays do. If you feel that way, you're not the only one. That's why there's a stage 2 TC39 proposal to add *Iterator Helpers* to the ECMAScript standard.

In the meantime, if you'd rather not write helpers yourself, you can find libraries that will help. Two popular examples include:

- *Itertools* (based on a popular Python library); and
- *IxJS* (like RxJS, but for iterables).

If you're after something more lightweight, you can take a look at my own personal toolkit, *Dynamo*. A word of warning though, it's written in TypeScript rather than the usual plain JavaScript I use here.

MESSAGE PASSING

The laziness of generators is fascinating and useful. But that's not all generators can do. Generators also allow us to pass messages in two directions between a pair of functions. Now, to be fair, functions can do this already. We pass a message to a *called* function by giving it parameters. And then the *calling* function receives a message back via the return value. But it's a one-shot thing. We get one message, one way, at the start. And one message, the other way, at the end. But generators allow us to send lots of messages back and forth.

The most commonly cited example of this is emulating / syntax. For example, suppose we're writing some Node code. The function we're writing needs to:

1. Read config from a file;
2. Make a fetch call to get an auth token; and
3. Make another fetch call with the token to get some data.

Using / , it might look like so:

But, before we had / , we could do something like this using generators:

And that looks *rather* like the / version, wouldn't you say? But, we need to do a bit of extra plumbing to make it work. And that extra plumbing looks like so:

This is neat. But perhaps not so useful. Most of us won't need to emulate / . But, that said, perhaps you use Babel (or similar) to make your code work with older browsers. If that's the case, transpiler will use generators like this to make / work for you. And, because generators are so low-level, we can tinker with this pattern in ways we can't with . This allows authors to create interesting libraries that go beyond / . For example, Kyle Simpson's *Cancellable Async Flows (CAF)*:

> *CAF [...] is a wrapper for generators that treats them like async functions, but with support for external cancellation via tokens. In this way, you can express flows of synchronous-looking asynchronous logic that are still cancelable (Cancelable Async Flows).*

Still, you may not need cancellable asynchronous processing (yet). Even so, there's more to generator message passing than working with promises. We can, for example, use them to simplify error handling.

I've written before about *handling errors using Either*. The Either structure lets us represent the result of an operation that might fail. By convention, we use a class called to represent a failure. And we use a class called to represent success. Here's a simplified version of the two Either classes:

Now, in that article, we ran through an example of parsing a CSV file. And we came up with a function like this for parsing a single row:

This is a function that returns an Either. But with generators and message passing, we can use to paper over the Eithers a little. It looks something like the following:

In this case, we expect the generator function to always yield Eithers. And the function works like so:

At the end of everything, we still get an Either value back from . And it's doing the same thing as . But this style of writing the code may feel more comfortable for some. Especially if they're not familiar with chained method calls.

WRAPPING UP

If you're not familiar with Generators, they may seem a little weird at first. And because they're so a low-level language, we can use them for *lots* of different applications. And that can make it difficult to see what you might want them for, day-to-day. But, as we've seen, they're useful for tasks like:

- Efficiently processing large data sets;
- Working with infinite sequences; and
- Passing messages between two functions.

Now, you may never feel the need to reach for generators. The kind of work you're doing might not suit them. But, it's still handy to have them in the toolbox, just in case. And if you start looking, you'll find generators working away behind the scenes in lots of places. And now and then, you might need to dig into some library code to work out what it's doing. In those cases, it's good to have an idea of how they work.

If looking at code in different ways interests you. Or, if you find yourself needing to adjust your coding style to meet your team's expectations…. I'm working on something I think you'll like. It's called *A Skeptic's Guide to Functional Programming with JavaScript*, and it's coming soon. Be sure to *subscribe* to learn more.

1. The coffee in Australia truly is wonderful. To understand how good Australians have it, picture this simple fact. In Australia, every McDonald's sells coffee made with full–sized espresso machines. And the machines are operated by trained baristas. Of course, it's McDonald's. Hence, the trained barista is often a fifteen–year–old earning minimum wage. And, as a consequence, the coffee is average. Coffee enthusiasts do not buy coffee from McDonald's. But, consider this:

McDonald's sets a baseline. If you want to start a café anywhere in Australia, you need to make coffee *at least* as good as McDonald's coffee. And the wonderful thing is, they do! ↩
2. From the description on Wikipedia: *https://en.wikipedia.org/wiki/Tim_Tam*. ↩
3. As described by Luke Henriques Gomes in *Australia's favourite choc bikkie, Tim Tams, takes on America*, *https://thenewdaily.com.au/life/eat-drink/2017/02/01/tim-tams-take-on-america/*. ↩

Subscribe via the electronic mail.